

EE49 Laboratory 3

Modulation

Due Date: 5/10/2011

After completing this lab assignment, you must be checked off by a TA in lab and provide your code via email. If you cannot make the allotted timeslots, it is your responsibility to setup an alternate appointment with the TAs before the due date. Good luck!

GOAL: The goal of this lab is to learn about modulation, and implement the wireless transmitter for a USRP-to-USRP link. Specifically we will be implementing the pulse shaping filter (root raised cosine) and the modulator (BPSK, QPSK).

Note: Most of the control traffic (especially error handling) has been handled for you.

1 Primer

(1) Modulation

As discussed in class, modulation is the process of varying some signal based upon a binary bitstream. In this case, we modulate the in-phase and quadrature components of a sinusoid to represent the information which we wish to send. We will utilize a technique called Phase Shift Keying (PSK). In PSK, the transmitter does not vary the amplitude or the frequency of the sinusoid being transmitted. Instead, it varies the phase according to the input bit sequence.

(2) Binary Phase Shift Keying (BPSK)

For example, BPSK represents each singular bit as a symbol, i.e. it transmits one of two possible phases corresponding to the bit value. For this assignment, binary 0 is mapped to a phase offset of $\pi/2$, i.e. $1j$ in complex notation. Binary 1 is mapped to a phase offset of $3\pi/2$, i.e. $-1j$ in complex notation. This **symbol mapping** is shown in figure 1. Mathematically, we represent the modulated signal for one bit as

$$s_b(t) = \cos(2\pi f_m t + \phi_b) \tag{1}$$

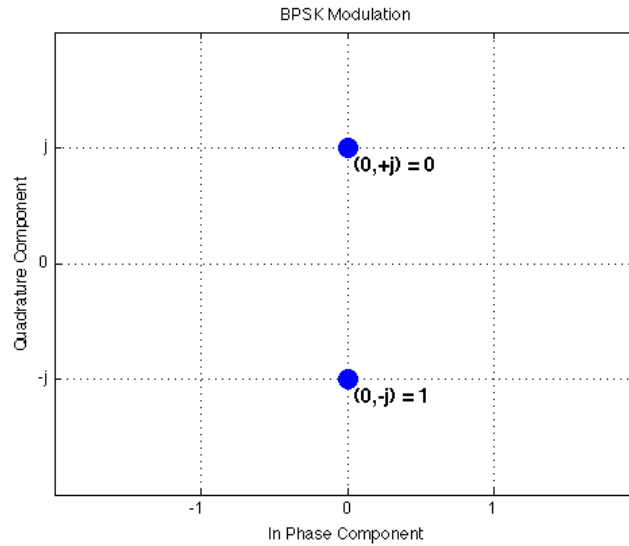


Figure 1: BPSK Symbol Map

where ϕ_b is the phase shift which represents $b = 0$ or 1 and f_m is the frequency of the baseband modulation. Generally, we choose f_m such that one full period of the sinusoid equals the symbol time. For our implementation, $\phi_0 = \frac{\pi}{2}$ and $\phi_1 = \frac{3\pi}{2}$. Using the equivalent I-Q notation, we can express our modulated signal $s_b(t)$ as

$$s_b(t) = \begin{cases} 0 \cos(2\pi f_m t) + j \sin(2\pi f_m t) & \text{if } b = 0 \\ 0 \cos(2\pi f_m t) - j \sin(2\pi f_m t) & \text{if } b = 1 \end{cases} \quad (2)$$

Can you see how this notation fits with the BPSK symbol map shown above?

If we wanted to send binary '101' using the mapping specified above, and the symbol time for each bit is equal to one full period of a sinusoid, the associated signal is shown in figure 2.

Make sure you're comfortable with this idea before proceeding. You might want to generate a short BPSK stream of your own in Matlab.

(3) Quadrature Phase Shift Keying (QPSK)

Similarly, in QPSK each *pair of 2 bits is mapped to a phase offset*. Thus, there are four possible phases that a QPSK transmitter will take. In this assignment, we choose the phase offsets $\pi/4, 3\pi/4, 5\pi/4, 7\pi/4$. In I-Q notation, these points are $0.707 + 0.707i, -0.707 + 0.707i, 0.707 - 0.707i, -0.707 - 0.707i$, respectively. This symbol map is shown in figure 3. To completely specify the mapping, we must decide

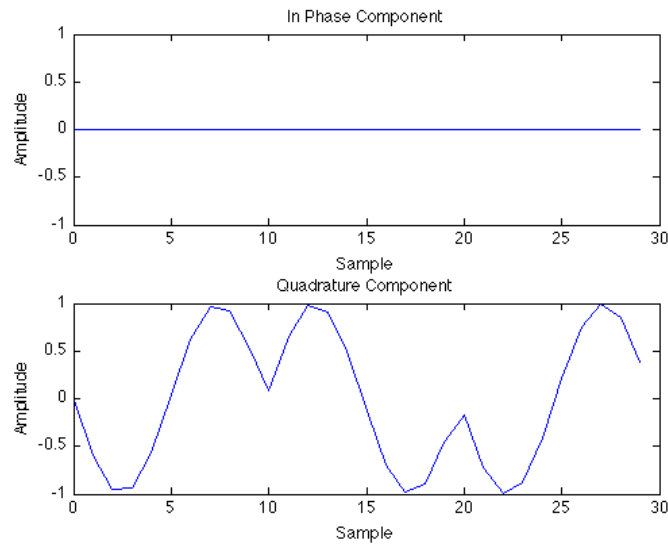


Figure 2: BPSK Stream

on an encoding, i.e. which sequence of 2 bits corresponds to which phase modulation. By far, the most common approach for this is **Gray Coding**.

The motivation for using Gray Coding is straightforward. Look at the plot showing the QPSK symbol map in figure 3. Notice that the symbols which are diagonal from each other are further apart than the symbols which are directly above/below or next to each other. Over communication channels with noise, sooner or later the receiver will make a mistake deciding which symbol was sent. For QPSK modulation, it's possible that we could choose a symbol in which **both** bits are guessed incorrectly, which is twice as bad as guessing only 1 bit incorrectly. In Gray coding, we map the bits such that the **nearest neighboring** symbols differ by only one bit. Thus, for the more likely incorrect guesses of the transmitted symbol, we make a mistake in only **one** bit. For this assignment, our gray code mapping is as follows

- 00 goes to $0.707 + 0.707i$
- 01 goes to $0.707 - 0.707i$
- 10 goes to $-0.707 + 0.707i$
- 11 goes to $-0.707 - 0.707i$

This symbol mapping is shown in figure 3.

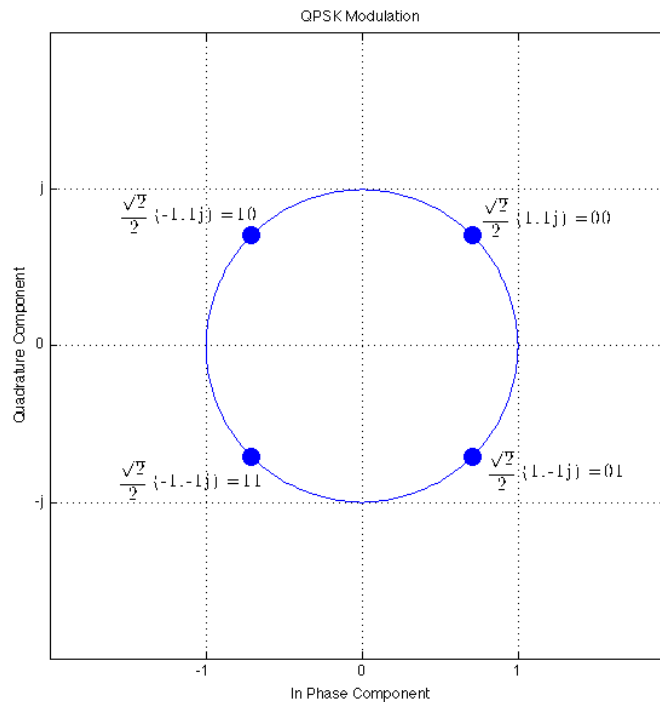


Figure 3: QPSK Modulation

(4) Filtering

Firstly recall from class that we wish to transmit and receive **band-limited** signals, i.e. our communication takes place over a limited frequency spectrum. Also, note from the figure 2, that a BPSK modulated stream would in general have sharp "edges" due to the sudden phase shifts, which leads to the generation of harmonic, higher frequency components. To avoid the spillover outside the frequency band, the transmit symbols are usually "filtered" to smooth the transitions and minimize the harmonic content. A good filter for such purposes is the "Root Raised Cosine Filter".

Recall from lecture, that passing a signal over a channel was equivalent to convolving the signal with the "Impulse Response" of the channel. Similarly, passing a signal through a filter is equivalent to convolving the signal with the time domain representation of the filter. However, unlike the channel, here the coefficients of the taps (denoted by $h[n]$) are in our control. These coefficients are so chosen so as to restrict the frequency component of the signal to a specified band. The coefficients of the taps for a "Root Raised Cosine" filter look like 4. This transmit side filter can also be referred to as a **Pulse Shaping Filter** because it gives a particular shape to the time domain representation of the signal we transmit.

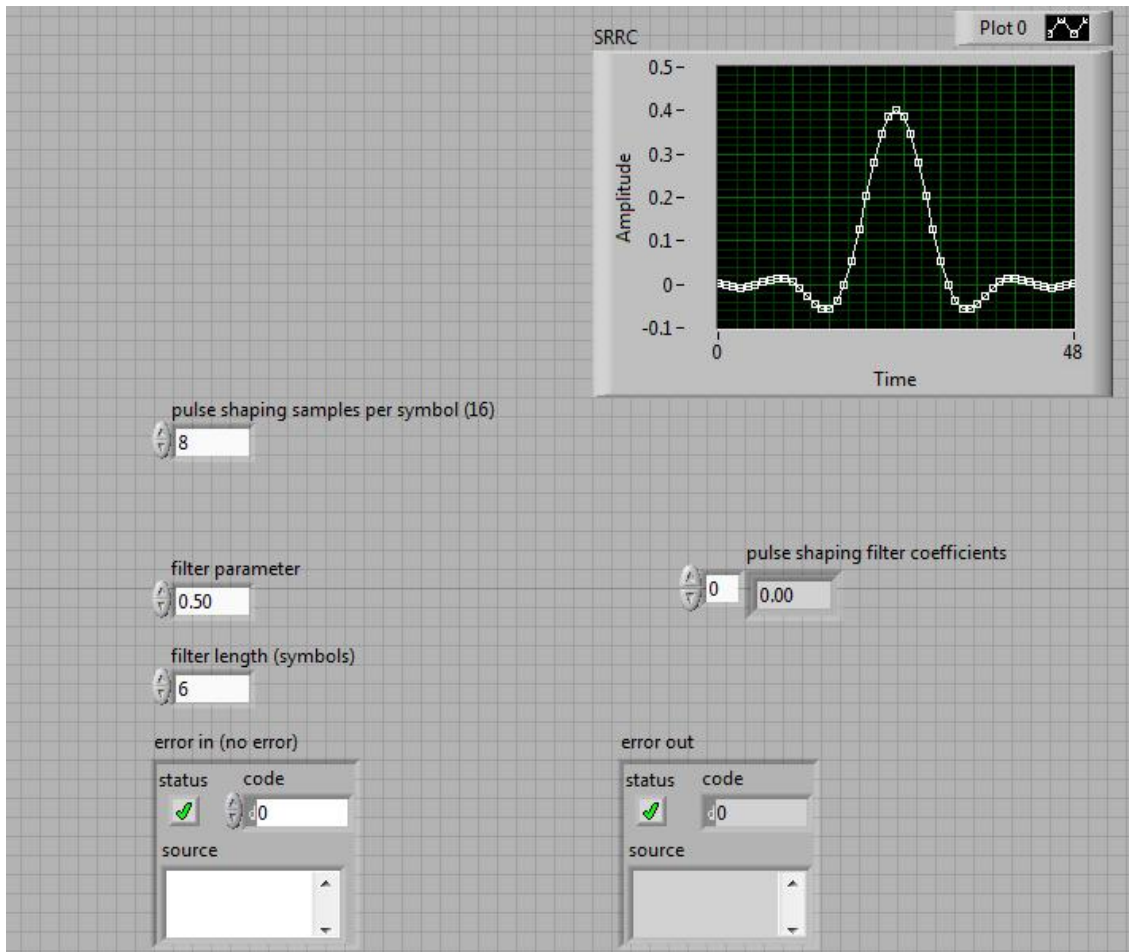


Figure 4: Time response of Root Raised Cosine Filter

2 Introduction

Open TX.vi, which consists of the basic transmitter side code. The basic structure of the code is in place - your job is to fill in the missing components. Essentially, this code flow is as follows:

1. Accept transmit (TX) parameters from the user via the front panel. These inputs include the type of modulation (BPSK or QPSK), the number of samples per symbol, and the parameters of the root raised cosine pulse shaping filter.
2. Based on the TX input parameters, the subVI "generate system parameters" generates the symbol map that will be used to map the input bitstream to mod-

ulated transmit symbols. It outputs a cluster named "PSK system parameters" which has 2 entries:

- A numeric "Samples per symbol"
 - An array of complex values "Symbol map"
3. The "PSK system parameters" and the "filter parameters" are passed on to the subVI "continuous PSK generation" as inputs. This subVI forms an array of randomly generated bits and modulates it using the symbol map generated in step (2) and then passes the resulting modulated signal through the pulse shaping filter.

Note that in this subVI, only a finite array of random bits is generated and hence the "output complex waveform" only consists of a finite array of complex transmit samples, this is referred to as a "**frame**". It is this frame which is repeated over and over again in the transmissions on the wireless channel as can be seen in the while loop in TX.vi. Note here that the subVI "niUSRP Write Tx data.vi" (which is being used inside the while loop) is a part of the USRP driver for LabVIEW and it interfaces the stream of samples generated to the USRP.

4. The front panel of TX.vi consists of the following controls and indicators:
- "Device names" is the IP address of the USRP. It should be set to the default value of "192.168.10.2".
 - The "filter parameters" is a control through which you can specify the pulse shaping filter parameters. The "TX Filter" we will be using is "Root Raised Cos". "Alpha" is a parameter of the filter called the roll off factor. "Filter length" defines the length of the filter in terms of the number of symbols. Default values of 0.5 for "Alpha" and 6 for "Filter Length" should work fine for our purposes.
 - Recall that a "frame" was a finite array of complex transmit samples which are repeated over and over again. Number of samples per "frame" is calculated and displayed on the front panel as the indicator "Frame Size[samples]".
 - "Choose a PSK format" is a control through which you can specify the modulation to use at the transmitter. The available options are QPSK and BPSK. Having chosen the modulation, you can also choose the number of "Samples per symbol" you want to transmit.
 - "TX parameters:" This is a control which sets the parameters of the USRP.
 - (a) "TX IQ Sampling Rate " is the number of samples per second that are transmitted by the USRP. Make sure this is set to the default value of 200k .

- (b) "TX Frequency" is the center frequency at which the transmitter transmits. Make sure this is set to the default value of 433.92MHz.
- (c) "TX Gain" gives you a control over the power at which to transmit. Typically for close range communication, the default value of 0 dB works fine.
- (d) "TX Antenna" is the antenna being used for transmission. Make sure this is set to "TX/RX".
- Note that having specified the "TX IQ Sampling rate" (samples per second transmitted) and the "Samples per symbol", their ratio automatically decides the "Symbol Rate [symbols/sec]" which is calculated and shown as an indicator on the front panel.

3 Task 1: Generate Symbol map

First open the "generate system parameters" subVI. Apart from the error handling, this subVI calls another subVI "generate PSK symbol map". This block only needs to form the "PSK system parameters" cluster as described in the Introduction.

Finish the block "generate PSK symbol map" by forming the symbol map according to the value of "M-PSK" ,i.e. depending on whether we need the symbol map for BPSK or QPSK. The symbol maps for both these constellations have been described in detail in the Primer.

- Inputs for "generate PSK symbol map"
 - "M-PSK", which is an integer corresponding to the number of points in the symbol map. Hence "M-PSK" would be 2 for BPSK and 4 for QPSK.
- Outputs for "generate PSK symbol map"
 - "bits per symbol" which would be 1 for BPSK and 2 for QPSK.
 - An array of complex numbers "Symbol Map" which holds the constellation diagram. The exact format you use is left upto you as long as you can interpret it later on (while doing the symbol mapping and generating symbols from the bitstream).

4 Task 2: Continuous PSK generation

The next task is to complete the "continuous PSK generation" subVI. Open this subVI to have a look at it. As described in the Introduction, this subVI generates an array of randomly generated bits and modulates it using the symbol map generated above and eventually applies the Pulse shaping filter.

Recall also that the "output complex waveform" only consists of a finite array of complex transmit samples, which is referred to as a "frame". It is this frame that is repeated over and over again over the wireless channel. Lets say we want each "frame" to consist of 1000 symbols.

There is some calculation done to evaluate the number of bits we would want to achieve 1000 symbols in each frame. This calculation will be partly explained later but is not essential for understanding the overall flow of the code.

Having calculated the number of bits we need to generate randomly, this value is passed on to the "random bit generator" subVI. Simultaneously, the subVI "validate and generate filter" generates the filter taps for a Root Raised Cosine Filter. Eventually, the filter taps and the random bit stream are both fed into the "Modulate PSK" subVI which performs the symbol mapping and applies the filter.

Description on the calculations being performed in "continuous PSK generation" :

As discussed above, lets assume we want the "frame" to consist of 1000 symbols. This implies the number of bits we need to generate would be $1000 \times (\text{bits per symbol})$ where "bits per symbol" is 1 for BPSK and 2 for QPSK. Recall that we would be eventually filtering this signal and as described in the primer, filtering is equivalent to convolution. We have seen in lecture and in lab2, that if the impulse response has k taps, then the first k output samples of the channel are not known even if the corresponding input samples are known due to "ISI" from previous unknown inputs. To account for this initialization of the filter, we generate $(1000 + \text{filter length}) \times (\text{bits per symbol})$ input bits. After symbol mapping, we would get $(1000 + \text{filter length}) \times (\text{samples per symbol})$ symbols. After the filtering, we would get $(1000 + \text{filter length}) \times (\text{samples per symbol})$ filtered samples and then we can delete the first $(\text{filter length}) \times (\text{samples per symbol})$ filtered samples to take care of ISI from previous unknown samples. Note here that the number of deleted samples is the same as the number of filter taps.

(1) Random Bit Generator

The subVI "random bit generator" generates an n length array of random bits. The array length is fed into the subVI as input. **Implement the random bit generation**

in "random bit generator.vi". Make sure that you generate a bit 0 and a bit 1 with almost equal likelihood.

- Inputs for "random bit generator"
 - "total bits" is the number of random bits we wish to generate.
- Outputs for "random bit generator"
 - "output bit stream" should be an array of length "total bits" consisting of 1s/0s representing the random bit stream generated.

(2) Generate filter

The subVI "validate and generate filter" first checks whether or not the inputs make sense. Afterwards, it generates the filter taps for the Root Raised Cosine filter. You do not need to modify this subVI. However, you can view its output and confirm that it looks like 4. Here, the output array "pulse shaping filter coefficients" is an array of the values of $h[n]$ corresponding to the Root raised cosine filter taps.

(3) Modulate PSK

Open "modulate PSK" and have a look at it. This has 2 major subVIs. The first one is "PSK map symbols". In this subVI, the array of incoming bits is mapped to symbols as described in the primer using the symbol map generated in "generate system parameters". **Implement this mapping in the subVI "PSK map symbols"**. That is, if for example, we are using a QPSK modulation and have an incoming bit sequence of 100 bits. Pick the first 2 bits and form the complex "symbol" according to the symbol map. Repeat this procedure until u generate 50 symbols from the 100 bits. Take care of corner cases where u might end up with an odd number of bits (just ignore the last bit in this case).

- Inputs for "PSK map symbols"
 - "Symbol Map" is the same array as generated in "generate PSKsymbol map"
 - "Input Bit Stream" is the same array of bits as generated in "random bit generator"
- Outputs for "PSK map symbols"
 - "Symbols" is an array of complex numbers representing the symbol stream resulting from the mapping of the bitstream.

(4) Filter PSK symbols

Eventually, these symbols are filtered in the subVI "filter PSK symbols" . Open this subVI and have a look. There is a lot of control traffic related to when you want to "reset" the USRP. All this has been done for you. Other than that, the major components of this subVI are "Upsampling.vi" and "perform convolution.vi". Note that the filter taps generated in "validate and generate filter" are passed on to this subVI as an input "Pulse Shaping Filter Coefficients" and the symbol stream generated in "PSK map symbols.vi" are passed on to this subVI as an input "Symbols".

Remember that the input "Symbols" is an array of all the symbols after the mapping. However, we wish to transmit multiple samples per symbol and then at the receiver decode the symbol from these multiple samples. Recall that in the lecture and in lab2, if (samples per symbol) were set to 4, we would repeat the symbol 4 times, i.e. if we wish to transmit the bitstream 101 we would transmit 4 samples of 5 Volts each to denote the first bit 1, then transmit 4 samples of 0 Volts each to denote the second bit 0 and finally transmit 4 samples of 5 volts each to denote the third bit 1.

Here we would be doing something different. We will first "upsample" the symbol stream which means we insert (samples per symbol - 1) zeros between every two symbols in the symbol stream. For example for the bitstream 101, the symbol stream would be '5volts 0volts 5volts' and the upsampled stream with (samples per symbol) set to 4 would be '5'000'0'000'5'000 where the samples in quotes are the actual symbols. **Carry out this upsampling in the subVI "upsampling.vi"**

- Inputs for "upsampling.vi"
 - "x" is an array of complex samples representing the stream that needs to be upsampled.
 - "L" is the amount of upsampling to be done, i.e. you must inset L-1 zeros between every two entries of "x".
- Outputs for "upsampling.vi"
 - "y" should be an array of complex samples representing the upsampled stream.

After upsampling, the convolution with filter taps is carried out. For this open the subVI "perform convolution.vi". Note that the filter taps are provided as the input "Pulse Shaping Filter Coefficients" which is an array whose first element is $h[0]$ and so on. The upsampled sample stream which needs to be filtered is the input "Sample stream". **Implement this convolution in the subVI "perform convolution.vi"**

- Inputs for "perform convolution.vi"
 - "Sample stream" is an array of complex numbers representing the input stream of samples that needs to be filtered.
 - "Pulse Shaping Filter Coefficients" is an array of complex numbers representing the values of the filter tap coefficients.
- Outputs for "perform convolution.vi"
 - "Filtered Samples" should be an array of complex numbers representing the filtered stream of samples, i.e. it must be the convolution of the "Sample stream" and the "Pulse Shaping Filter Coefficients".

Note: In the upsampling and convolution blocks, keep in mind that we are dealing with complex numbers and hence ensure that all constants and variables you introduce are defined to be complex. You can change the "Representation" of a constant/variable by right clicking on it

5 Testing the code

You will test the code against a receiver and see whether the receiver decodes the constellation. 5 and 6 are screenshots of the TX and RX front panels when the code is working.

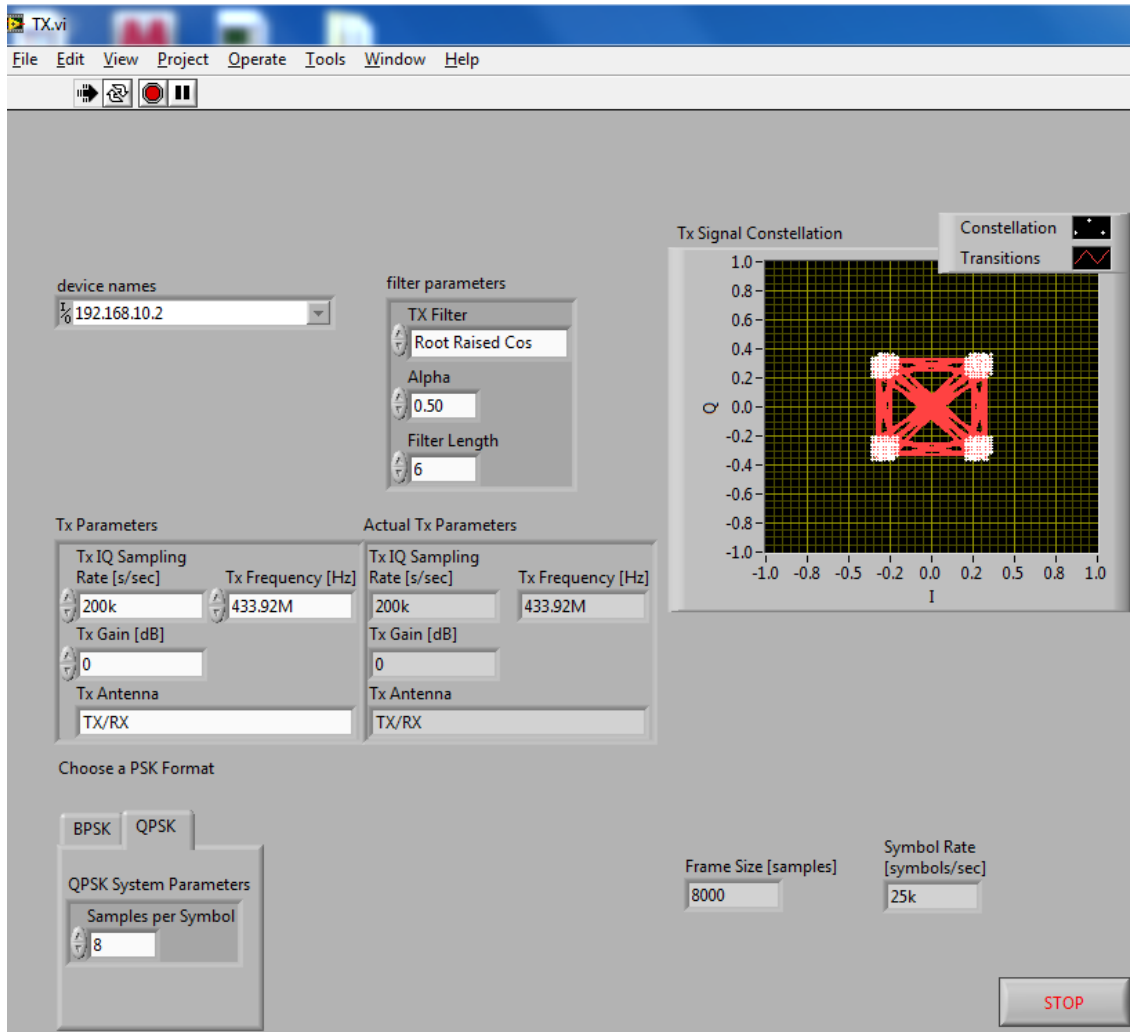


Figure 5: Screenshot of a working code (TX)

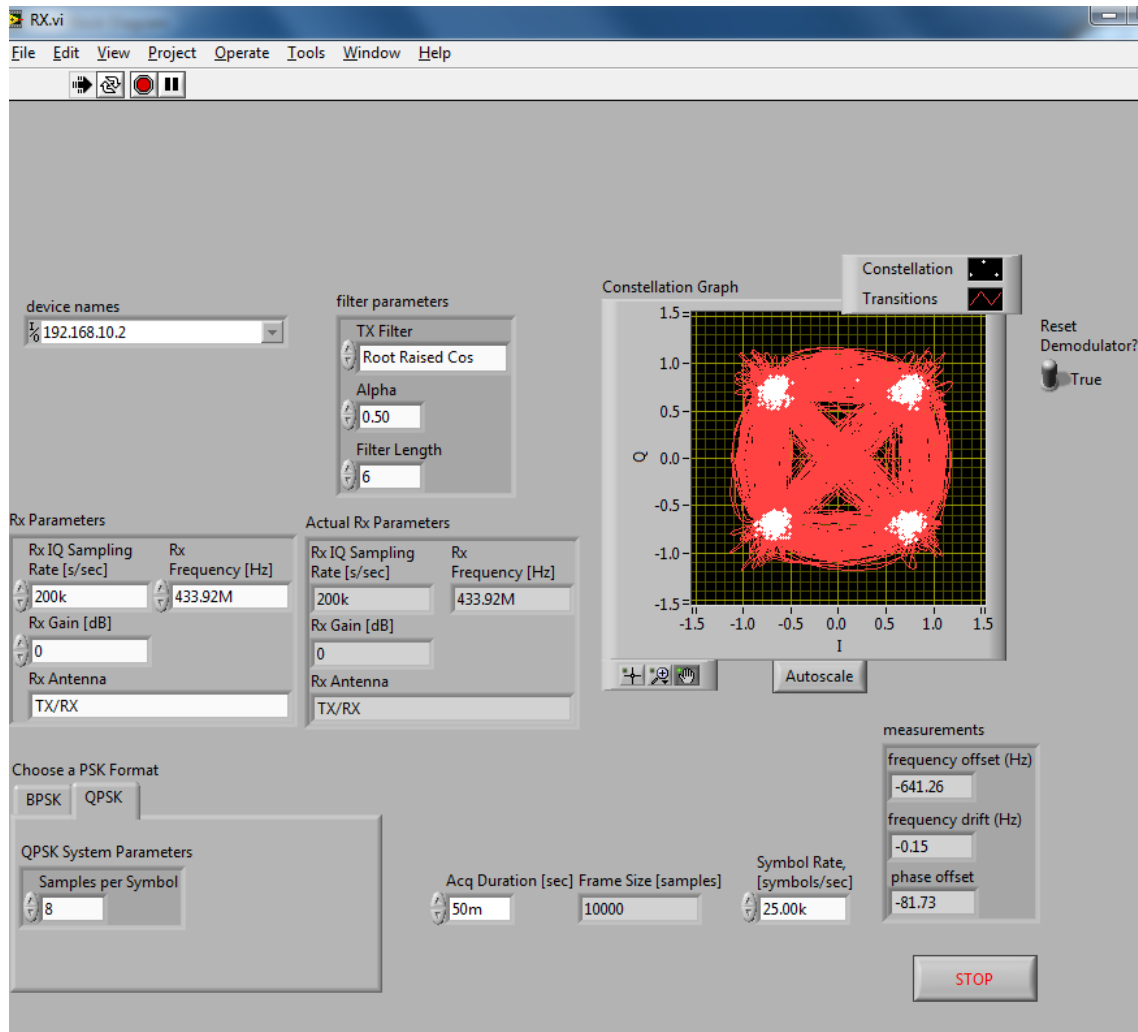


Figure 6: Screenshot of a working code (RX)